BLOCKSEC

Security Audit Report for Perp Contracts

Date: May 18, 2026 **Version:** 1.0

Contact: contact@blocksec.com

Contents

Report Manifest

Item	Description
Client	HertzFlow
Target	Perp Contracts

Version History

Version	Date	Description
1.0	May 18, 2026	First release

Signature

Digitally signed by: BlockSec (https://blocksec.com)

SHA1 digest of public key: BAB106D3708BC6DE0963CE05E9478C24B326AE1C

Date: 2026-05-26 09:42:06+0000

About BlockSec BlockSec focuses on the security of the blockchain ecosystem and collaborates with leading DeFi projects to secure their products. BlockSec is founded by top-notch security researchers and experienced experts from both academia and industry. They have published multiple blockchain security papers in prestigious conferences, reported several zero-day attacks of DeFi applications, and successfully protected digital assets that are worth more than 14 million dollars by blocking multiple attacks. They can be reached at Email, Twitter and Medium.

Chapter 1 Introduction

1.1 About the Audit Target

Information	Description
Type	Smart Contract
Language	Solidity
Approach	Semi-automatic and manual verification

The audit target (hereinafter referred to as the Target) of this audit is the code repository [1] of Perp Contracts of HertzFlow.

The Perp Contracts of HertzFlow is a perpetual exchange forked from GMX Synthetics v2.3-branch. It implements the Zero Funding Position (i.e., ZFP) feature, enabling ultra high leverage trading (e.g., 75x to 555x). Unlike normal positions that charge position fees, ZFP positions replace fees with a kink model profit extraction mechanism where the protocol takes a portion of profits based on ROI tiers, with higher ROI yielding more favorable user keep ratios. To support the ZFP feature safely, the project introduces several mechanisms. Specifically, a collateral range check (i.e., willBeSufficient and willBeTooMuch) is implemented for all ZFP positions to ensure their validity after the corresponding operations (e.g., position decrease). Additionally, an Early Force Liquidation (i.e., EFL) feature is implemented to forcibly close positions with excessive profits to protect liquidity providers.

Note this audit only focuses on the smart contracts in the following directories/files. Code prior to and including the baseline version 0, where applicable, is outside the scope of this audit and assumes to be reliable and secure.

- contracts/adl/AdlUtils.sol
- contracts/callback/CallbackUtils.sol
- contracts/callback/IHlvDepositCallbackReceiver.sol
- contracts/callback/IHlvWithdrawalCallbackReceiver.sol
- contracts/config/Config.sol
- contracts/config/ConfigUtils.sol
- contracts/data/Keys.sol
- contracts/data/Keys2.sol
- contracts/deposit/Deposit.sol
- contracts/deposit/ExecuteDepositUtils.sol
- contracts/error/Errors.sol
- contracts/exchange/AdlHandler.sol
- contracts/exchange/BaseHandler.sol
- contracts/exchange/HlvDepositHandler.sol
- contracts/exchange/HlvShiftHandler.sol
- contracts/exchange/HlvWithdrawalHandler.sol
- contracts/exchange/IHlvDepositHandler.sol

[1] https://github.com/HertzFlow/contracts-v2-bsc-audit

- contracts/exchange/IHlvWithdrawalHandler.sol
- contracts/exchange/IOrderHandler.sol
- contracts/exchange/LiquidationHandler.sol
- contracts/fee/FeeHandler.sol
- contracts/gas/GasUtils.sol
- contracts/hlv/
- contracts/liquidation/EarlyForceLiquidationUtils.sol
- contracts/liquidation/LiquidationUtils.sol
- contracts/market/MarketFactory.sol
- contracts/market/MarketToken.sol
- contracts/market/MarketUtils.sol
- contracts/market/PositionImpactPoolUtils.sol
- contracts/order/BaseOrderUtils.sol
- contracts/order/DecreaseOrderUtils.sol
- contracts/order/IBaseOrderUtils.sol
- contracts/order/IncreaseOrderUtils.sol
- contracts/order/Order.sol
- contracts/order/OrderEventUtils.sol
- contracts/order/OrderStoreUtils.sol
- contracts/order/OrderUtils.sol
- contracts/position/DecreasePositionCollateralUtils.sol
- contracts/position/DecreasePositionUtils.sol
- contracts/position/DecreaseZFPPositionCollateralUtils.sol
- contracts/position/DecreaseZFPPositionUtils.sol
- contracts/position/IncreasePositionUtils.sol
- contracts/position/Position.sol
- contracts/position/PositionEventUtils.sol
- contracts/position/PositionStoreUtils.sol
- contracts/position/PositionUtils.sol
- contracts/position/ZFPPositionUtils.sol
- contracts/pricing/PositionPricingUtils.sol
- contracts/reader/HlvReader.sol
- contracts/reader/ReaderPositionUtils.sol
- contracts/router/ExchangeRouter.sol
- contracts/router/HlvRouter.sol
- contracts/router/SubaccountRouter.sol
- contracts/router/relay/BaseGelatoRelayRouter.sol
- contracts/router/relay/IRelayUtils.sol
- contracts/router/relay/RelayUtils.sol
- contracts/shift/ShiftUtils.sol
- contracts/utils/Array.sol
- contracts/withdrawal/Withdrawal.sol

Other files are not within the scope of the audit. Additionally, all dependencies of the

Target are considered reliable in terms of both functionality and security, and are therefore not included in the audit scope.

The auditing process is iterative. Specifically, we would audit the commits that fix the discovered issues. If there are new issues, we will continue this process. The commit SHA values during the audit are shown in the following table. Our audit report is responsible for the code in the initial version (`Version 1`), as well as new code (in the following versions) to fix issues in the audit report. Code prior to and including the baseline version (`Version 0` [2]), where applicable, is outside the scope of this audit and assumes to be reliable and secure.

Project	Version	Commit Hash
Perp Contracts	`Version 0`	`1cc4ef9ca19f26bd51b3e175d2c0d3a595a2ac02`
	`Version 1`	`46477c0105fdd37a92e4d6e1ec96ee7fcdfcfda7`
	`Version 2`	`860811ca56a51529cd969c46b62e283376cbfd6e`

1.2 Disclaimer

This audit report does not constitute investment advice or a personal recommendation. It does not consider, and should not be interpreted as considering or having any bearing on, the potential economics of a token, token sale or any other product, service or other asset. Any entity should not rely on this report in any way, including for the purpose of making any decisions to buy or sell any token, product, service or other asset.

This audit report is not an endorsement of any particular project or team, and the report does not guarantee the security of any particular project. This audit does not give any warranties on discovering all security issues of the Target, i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues. As one audit cannot be considered comprehensive, we always recommend proceeding with independent audits and a public bug bounty program to ensure the security of the Target.

The scope of this audit is limited to the code mentioned in Section 1.1. Unless explicitly specified, the security of the language itself (e.g., the solidity language), the underlying compiling toolchain and the computing infrastructure are out of the scope.

1.3 Procedure of Auditing

We perform the audit according to the following procedure.

- **Vulnerability Detection** We first scan the Target with automatic code analyzers, and then manually verify (reject or confirm) the issues reported by them.
- **Semantic Analysis** We study the business logic of the Target and conduct further investigation on the possible vulnerabilities using an automatic fuzzing tool (developed by our research team). We also manually analyze possible attack scenarios with independent auditors to cross-check the result.
- **Recommendation** We provide some useful advice to developers from the perspective of good programming practice, including gas optimization, code style, and etc.

[2] The `Version 0`'s commit hash can be found in the GMX Synthetics V2 repository.

We show the main concrete checkpoints in the following.

1.3.1 Security Issues

* Access control
* Permission management
* Whitelist and blacklist mechanisms
* Initialization consistency
* Improper use of the proxy system
* Reentrancy
* Denial of Service (DoS)
* Untrusted external call and control flow
* Exception handling
* Data handling and flow
* Events operation
* Error-prone randomness
* Oracle security
* Business logic correctness
* Semantic and functional consistency
* Emergency mechanism
* Economic and incentive impact

1.3.2 Additional Recommendation

* Gas optimization
* Code quality and style

Note *The previous checkpoints are the main ones. We may use more checkpoints during the auditing process according to the functionality of the project.*

1.4 Security Model

To evaluate the risk, we follow the standards or suggestions that are widely adopted by both industry and academy, including OWASP Risk Rating Methodology [3] and Common Weakness Enumeration [4]. The overall *severity* of the risk is determined by *likelihood* and *impact*. Specifically, likelihood is used to estimate how likely a particular vulnerability can be uncovered and exploited by an attacker, while impact is used to measure the consequences of a successful exploit.

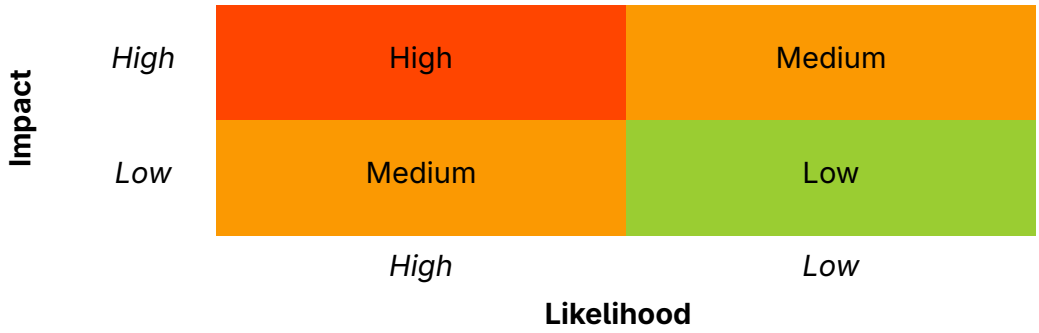
In this report, both likelihood and impact are categorized into two ratings, i.e., *high* and *low* respectively, and their combinations are shown in Table 1.1.

Accordingly, the severity measured in this report are classified into three categories: **High**, **Medium**, **Low**. For the sake of completeness, **Undetermined** is also used to cover circumstances when the risk cannot be well determined.

[3] https://owasp.org/www-community/OWASP_Risk_Rating_Methodology

[4] https://cwe.mitre.org/

Table 1.1: Vulnerability Severity Classification



Furthermore, the status of a discovered item will fall into one of the following five categories:

- **Undetermined** No response yet.
- **Acknowledged** The item has been received by the client, but not confirmed yet.
- **Confirmed** The item has been recognized by the client, but not fixed yet.
- **Partially Fixed** The item has been confirmed and partially fixed by the client.
- **Fixed** The item has been confirmed and fixed by the client.

Chapter 2 Findings

In total, we found **five** potential security issues. Besides, we have **two** recommendations and **seven** notes.

- High Risk: 3
- Medium Risk: 2
- Recommendation: 2
- Note: 7

ID	Severity	Description	Category	Status
1	High	ZFP position fee circumvention due to the ROI manipulation	Security Issue	Fixed
2	High	Insufficient collateral checks for ZFP positions in the `IncreasePositionUtils` contract	Security Issue	Fixed
3	High	Lack of availability checks for ZFP-related orders during order executions	Security Issue	Fixed
4	Medium	Incorrect ZFP position-size checks in the function `decreaseZFPPosition()`	Security Issue	Fixed
5	Medium	Ineffective validations for ZFP-specific configurations in the `validateRange()` function	Security Issue	Fixed
6	-	Revise the insufficient range checks in the function `validateRange()`	Recommendation	Confirmed
7	-	Revise the market token symbol	Recommendation	Fixed
8	-	Proper position size delta configuration when applying ADL operations to ZFP positions	Note	-
9	-	Proper priority between EFL and ADL operations	Note	-
10	-	Risks for supporting RWA and commodity markets	Note	-
11	-	Potential HLV price inflation due to EFL operations	Note	-
12	-	Ensure gasless order creation entry points are disabled	Note	-
13	-	Potential centralization risks	Note	-
14	-	Deprecated modules and logic	Note	-

The details are provided in the following sections.

2.1 Security Issue

2.1.1 ZFP position fee circumvention due to the ROI manipulation

Severity High

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the contract `DecreaseZFPPositionCollateralUtils`, the `decreaseZFPPosition()` function extracts fees from a ZFP position's profit by invoking the function `extractZFPProfit()`. The extracted amount of profit depends on the ZFP position's ROI. Specifically, in the profit extraction mechanism, a higher ROI yields a higher keep ratio, letting users retain more profit. However, this design is vulnerable to ROI manipulation. Users can perform collateral-only withdrawals (i.e., decrease orders with `sizeDeltaUsd == 0`) to artificially inflate ROI before closing their ZFP positions. This manipulation moves the ZFP position in a higher keep ratio segment without decreasing actual profit, enabling users to retain more profit. As a result, the protocol and LPs may lose fees due to the ROI manipulation.

```
223         (
224             PositionUtils.DecreasePositionCollateralValues memory values,
225             PositionPricingUtils.PositionFees memory fees
226         ) = DecreaseZFPPositionCollateralUtils.processZFPCollateral(
227             params,
228             cache
229         );
230         // NOTE: @kayce extrat profit from zfp position
231         (values, /*extractedProfit*/) = ZFPPositionUtils.extractZFPProfit(
232             params,
233             values,
234             cache.collateralTokenPrice,
235             cache.pnlToken
236         );
```

Listing 2.1: contracts/position/DecreaseZFPPositionUtils.sol

```
52  function processZFPCollateral(
53      PositionUtils.UpdatePositionParams memory params,
54      PositionUtils.DecreasePositionCache memory cache
55  ) external returns (
56      PositionUtils.DecreasePositionCollateralValues memory,
57      PositionPricingUtils.PositionFees memory
58  ) {
59      ProcessCollateralCache memory collateralCache;
60      PositionUtils.DecreasePositionCollateralValues memory values;
61
62      values.output.outputToken = params.position.collateralToken();
63      values.output.secondaryOutputToken = cache.pnlToken;
64
65      // only allow insolvent closing if it is a liquidation or ADL order
66      // isInsolventCloseAllowed is used in handleEarlyReturn to determine
67      // whether the txn should revert if the remainingCostUsd is below zero
```

```
68      //
69      // for isInsolventCloseAllowed to be true, the sizeDeltaUsd must equal
70      // the position size, otherwise there may be pending positive pnl that
71      // could be used to pay for fees and the position would be undercharged
72      // if the position is not fully closed
73      //
74      // for ADLs it may be possible that a position needs to be closed by a larger
75      // size to fully pay for fees, but closing by that larger size could cause a
            PnlOvercorrected
76      // error to be thrown in AdlHandler, this case should be rare
77      collateralCache.isInsolventCloseAllowed =
78          params.order.sizeDeltaUsd() == params.position.sizeInUsd() &&
79          (
80              Order.isLiquidationType(params.order.orderType()) ||
81              params.secondaryOrderType == Order.SecondaryOrderType.Adl
82          );
83
84      (values.priceImpactUsd, values.executionPrice, collateralCache.balanceWasImproved) =
            PositionUtils.getExecutionPriceForDecrease(params, cache.prices.indexTokenPrice);
85
86      // the totalPositionPnl is calculated based on the current indexTokenPrice instead of the
            executionPrice
87      // since the executionPrice factors in price impact which should be accounted for
            separately
88      // the sizeDeltaInTokens is calculated as position.sizeInTokens() * sizeDeltaUsd / position
            .sizeInUsd()
89      // the basePnlUsd is the pnl to be realized, and is calculated as:
90      // totalPositionPnl * sizeDeltaInTokens / position.sizeInTokens()
91      (values.basePnlUsd, values.uncappedBasePnlUsd, values.sizeDeltaInTokens) = PositionUtils.
            getPositionPnlUsd(
92          params.contracts.dataStore,
93          params.market,
94          cache.prices,
95          params.position,
96          params.order.sizeDeltaUsd()
97      );
```

Listing 2.2: contracts/position/DecreaseZFPPositionCollateralUtils.sol

```
23
24  function extractZFPProfit(
25      PositionUtils.UpdatePositionParams memory params,
26      PositionUtils.DecreasePositionCollateralValues memory values,
27      Price.Props memory collateralTokenPrice,
28      address pnlToken
29  ) internal returns (PositionUtils.DecreasePositionCollateralValues memory, uint256) {
30      // NOTE: value.output.outputAmount at here is basePnL + positive price impact fee - other
            fees
31      // for hertzflow, only USDT-USDT pools exist, so the pnl and collateral token must be USDT.
32
33      // no profit to extract
34      // basePnlUsd < 0 means the position lost
35      // outputAmount == 0 means there's nothing left for us after paying cost
```

```
36      if (values.canBeExtractedProfit == 0 || values.basePnlUsd <= 0) {
37          return (values, 0);
```

Listing 2.3: contracts/position/ZFPPositionUtils.sol

Impact The protocol and LPs may lose fees due to the ROI manipulation.

Suggestion Revise the code accordingly.

Clarification from BlockSec The project fixed the issue by introducing a proportional decrease limit to prevent collateral-only decrease for ZFP positions. It is also recommended to enforce proportional decrease limits at decrease-order creation.

2.1.2 Insufficient collateral checks for ZFP positions in the `IncreasePositionUtils` contract

Severity High

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the contract `IncreasePositionUtils`, the function `increasePosition()` handles position and collateral adjustments. To support ZFP positions, this function implements an additional collateral check (Line 198) that enforces the required leverage range (e.g., 75x - 555x) for ZFP positions. However, this check runs only when the position size is increased (i.e., `params.order.sizeDeltaUsd() > 0`). As a result, depositing collateral without changing position size circumvents the additional check, allowing the ZFP position to violate the minimum leverage constraint (i.e., 75x), breaking the design invariant.

```
172     if (params.order.sizeDeltaUsd() > 0) {
173         // reserves are only validated if the sizeDeltaUsd is more than zero
174         // this helps to ensure that deposits of collateral into positions
175         // should still succeed even if pool tokens are fully reserved
176         MarketUtils.validateReserve(
177             params.contracts.dataStore,
178             params.market,
179             prices,
180             params.order.isLong()
181         );
182
183         MarketUtils.validateOpenInterestReserve(
184             params.contracts.dataStore,
185             params.market,
186             prices,
187             params.order.isLong()
188         );
189
190         PositionUtils.WillPositionCollateralBeSufficientValues memory positionValues =
                PositionUtils.WillPositionCollateralBeSufficientValues(
191             params.position.sizeInUsd(), // positionSizeInUsd
192             params.position.collateralAmount(), // positionCollateralAmount
193             0,  // realizedPnlUsd
194             0 // openInterestDelta
```

```
195         );
196
197         if (params.position.isZFP()) {
198             (bool willBeSufficient, bool willBeTooMuch, int256 remainingCollateralUsd) =
                    PositionUtils.willZFPPositionCollateralBeSufficient(
199                 params.contracts.dataStore,
200                 params.market,
201                 prices,
202                 params.position.collateralToken(),
203                 positionValues
204             );
205
206             if (!willBeSufficient) {
207                 revert Errors.InsufficientCollateralUsd(remainingCollateralUsd);
208             }
209             if (willBeTooMuch) {
210                 revert Errors.TooMuchCollateralUsdForZFP(remainingCollateralUsd);
211             }
```

Listing 2.4: contracts/position/IncreasePositionUtils.sol

Impact ZFP positions may circumvent the minimum leverage constraint, breaking the intentional design.

Suggestion Apply the collateral check when users only increase collateral for their ZFP positions.

2.1.3 Lack of availability checks for ZFP-related orders during order executions

Severity High

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the contract `OrderUtils`, the function `createOrder()` checks the ZFP availability via the function `validateMarketZFPEnabled()`. However, the availability check is missing during order execution (i.e., the function `processOrder()` in both contracts `DecreaseOrderUtils` and `IncreaseOrderUtils`). As a result, ZFP-related orders can be executed even when the ZFP feature is disabled, exposing the protocol to potential risks.

```
128     if (Order.isPositionOrder(params.orderType)) {
129         MarketUtils.validatePositionMarket(dataStore, params.addresses.market);
130         // NOTE: @kayce validate if zfp enabled on this market if the order is a zfp order
131         if (params.orderPositionType == Order.OrderPositionType.ZFP) {
132             MarketUtils.validateMarketZFPEnabled(dataStore, params.addresses.market);
133         }
```

Listing 2.5: contracts/order/OrderUtils.sol

```
18  function processOrder(BaseOrderUtils.ExecuteOrderParams memory params) internal returns (
        EventUtils.EventLogData memory) {
19      MarketUtils.validatePositionMarket(params.contracts.dataStore, params.market);
20
```

```
21      (address collateralToken, uint256 collateralIncrementAmount) = params.contracts.swapHandler
            .swap(ISwapUtils.SwapParams(
22          params.contracts.dataStore,
23          params.contracts.eventEmitter,
24          params.contracts.oracle,
25          params.contracts.orderVault,
26          params.key,
27          params.order.initialCollateralToken(),
28          params.order.initialCollateralDeltaAmount(),
29          params.swapPathMarkets,
30          params.order.minOutputAmount(),
31          params.order.market(),
32          params.order.uiFeeReceiver(),
33          false,
34          ISwapPricingUtils.SwapPricingType.Swap
35      ));
36
37      MarketUtils.validateMarketCollateralToken(params.market, collateralToken);
38
39      Order.OrderPositionType positionType = params.order.orderPositionType();
40
41      bytes32 positionKey = BaseOrderUtils.getPositionKey(params.order.account(), params.order.
            market(), collateralToken, params.order.isLong(), positionType);
42      Position.Props memory position = PositionStoreUtils.get(params.contracts.dataStore,
            positionKey);
```

Listing 2.6: contracts/order/IncreaseOrderUtils.sol

```
30  function processOrder(BaseOrderUtils.ExecuteOrderParams memory params) internal returns (
        EventUtils.EventLogData memory) {
31      Order.Props memory order = params.order;
32      MarketUtils.validatePositionMarket(params.contracts.dataStore, params.market);
33
34      Order.OrderPositionType positionType = order.orderPositionType();
35
36      bytes32 positionKey = BaseOrderUtils.getPositionKey(order.account(), order.market(), order.
            initialCollateralToken(), order.isLong(), positionType);
37      Position.Props memory position = PositionStoreUtils.get(params.contracts.dataStore,
            positionKey);
38      PositionUtils.validateNonEmptyPosition(position);
39
40      validateOracleTimestamp(
41          params.contracts.dataStore,
42          order.orderType(),
43          order.updatedAtTime(),
44          order.validFromTime(),
45          position.increasedAtTime(),
46          position.decreasedAtTime(),
47          params.minOracleTimestamp,
48          params.maxOracleTimestamp
49      );
50
51      PositionUtils.UpdatePositionParams memory positionParams = PositionUtils.
```

```
            UpdatePositionParams(
52          params.contracts,
53          params.market,
54          order,
55          params.key,
56          position,
57          positionKey,
58          params.secondaryOrderType
59      );
60
61      DecreasePositionUtils.DecreasePositionResult memory result = positionType == Order.
            OrderPositionType.ZFP
62          ? DecreaseZFPPositionUtils.decreaseZFPPosition(positionParams)
63          : DecreasePositionUtils.decreasePosition(positionParams);
```

Listing 2.7: contracts/order/DecreaseOrderUtils.sol

Impact ZFP-related orders can be executed even when the ZFP feature is disabled.

Suggestion Add the availability checks (i.e., the function `validateMarketZFPEnabled()`) during the order execution.

Clarification from BlockSec The project must ensure keepers correctly handle both ZFP and normal positions during ADL when disabling the ZFP feature or the market.

2.1.4 Incorrect ZFP position-size checks in the function `decreaseZFPPosition()`

Severity Medium

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the contract `DecreaseZFPPositionUtils`, the function `decreaseZFPPosition()` handles decreasing a ZFP position. In this function, a position-size check (Line 165) is performed to ensure that ZFP positions are auto-closed when their size falls below the minimum value. However, the check incorrectly compares the remaining position size against `MIN_POSITION_SIZE_USD` instead of `MIN_ZFP_POSITION_SIZE_USD`. As a result, ZFP positions cannot be auto-closed due to the incorrect comparison.

```
163         if (
164             params.position.sizeInUsd() > params.order.sizeDeltaUsd() &&
165             params.position.sizeInUsd() - params.order.sizeDeltaUsd() < params.contracts.
                dataStore.getUint(Keys.MIN_POSITION_SIZE_USD)
166         ) {
```

Listing 2.8: contracts/position/DecreaseZFPPositionUtils.sol

Impact ZFP positions with invalid position size cannot be auto-closed, breaking the intentional design.

Suggestion Correct the position-size check in the function `decreaseZFPPosition()`.

2.1.5 Ineffective validations for ZFP-specific configurations in the `validateRange()` function

Severity Medium

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the contract `ConfigUtils`, the function `validateRange()` checks whether provided configuration values fall within the allowed range. However, ZFP-specific validations use the base key to fetch values for comparison instead of the full key (i.e., `getFullKey(baseKey, market)`). As a result, the incorrect key usage can render validations ineffective for ZFP-specific configurations (e.g., `ZFP_PROFIT_EXTRACT_OPTIMAL_ROI_*`, `MAX_ZFP_COLLATERAL_FACTOR`).

```
372             baseKey == Keys.ZFP_PROFIT_EXTRACT_OPTIMAL_ROI_1
373         ) {
374             if (dataStore.getUint(Keys.ZFP_PROFIT_EXTRACT_OPTIMAL_ROI_0) > value) {
375                 revert Errors.ConfigValueExceedsAllowedRange(baseKey, value);
376             }
377         }
378         if (
379             baseKey == Keys.ZFP_PROFIT_EXTRACT_OPTIMAL_ROI_2
380         ) {
381             if (dataStore.getUint(Keys.ZFP_PROFIT_EXTRACT_OPTIMAL_ROI_1) > value) {
382                 revert Errors.ConfigValueExceedsAllowedRange(baseKey, value);
383             }
384         }
385         if (
386             baseKey == Keys.ZFP_PROFIT_EXTRACT_OPTIMAL_ROI_3
387         ) {
388             if (dataStore.getUint(Keys.ZFP_PROFIT_EXTRACT_OPTIMAL_ROI_2) > value) {
389                 revert Errors.ConfigValueExceedsAllowedRange(baseKey, value);
390             }
391         }
392         if (
393             baseKey == Keys.ZFP_PROFIT_EXTRACT_OPTIMAL_ROI_4
394         ) {
395             if (dataStore.getUint(Keys.ZFP_PROFIT_EXTRACT_OPTIMAL_ROI_3) > value) {
396                 revert Errors.ConfigValueExceedsAllowedRange(baseKey, value);
397             }
398         }
```

Listing 2.9: contracts/config/ConfigUtils.sol

```
407             baseKey == Keys.MAX_ZFP_COLLATERAL_FACTOR
408         ) {
409             if (dataStore.getUint(Keys.MIN_ZFP_COLLATERAL_FACTOR) > value) {
410                 revert Errors.ConfigValueExceedsAllowedRange(baseKey, value);
411             }
412         }
413         if (
```

Listing 2.10: contracts/config/ConfigUtils.sol

Impact The incorrect key usage can render validations ineffective for ZFP-specific configurations in the function `validateRange()`.

Suggestion Use the full key in the ZFP-specific configuration validation.

2.2 Recommendation

2.2.1 Revise the insufficient range checks in the function `validateRange()`

Status Confirmed

Introduced by `Version 1`

Description In the contract `ConfigUtils`, the function `validateRange()` validates whether provided configuration values fall within allowed ranges. However, several checks in this function are insufficient. It is recommended to revise the following validation to improve robustness.

1. The value of `MIN_ZFP_COLLATERAL_FACTOR_FOR_LIQUIDATION` is not validated. It is recommended to add a check to ensure that the value of `MIN_ZFP_COLLATERAL_FACTOR_FOR_LIQUIDATION` is less than the value of `MIN_ZFP_COLLATERAL_FACTOR`.

```
498       if (
499           baseKey == Keys.FUNDING_FACTOR ||
500           baseKey == Keys.BORROWING_FACTOR ||
501           baseKey == Keys.FUNDING_INCREASE_FACTOR_PER_SECOND ||
502           baseKey == Keys.FUNDING_DECREASE_FACTOR_PER_SECOND ||
503           baseKey == Keys.MIN_COLLATERAL_FACTOR_FOR_LIQUIDATION
504       ) {
505           // revert if value > 1%
506           if (value > 1 * Precision.FLOAT_PRECISION / 100) {
507               revert Errors.ConfigValueExceedsAllowedRange(baseKey, value);
508           }
509       }
```

Listing 2.11: contracts/config/ConfigUtils.sol

```
462       allowedBaseKeys[Keys.MIN_COLLATERAL_FACTOR_FOR_LIQUIDATION] = true;
463       allowedBaseKeys[Keys.MIN_ZFP_COLLATERAL_FACTOR_FOR_LIQUIDATION] = true;
```

Listing 2.12: contracts/config/Config.sol

2. There are no overlap checks for ROI-related values (e.g., `ZFP_PROFIT_EXTRACT_OPTIMAL_ROI_1`). Specifically, the current validations do not ensure that ROI thresholds are strictly monotonically increasing.

```
365   function validateRange(
366       DataStore dataStore,
367       bytes32 baseKey,
368       bytes memory data,
369       uint256 value
370   ) public view {
371       if (
372           baseKey == Keys.ZFP_PROFIT_EXTRACT_OPTIMAL_ROI_1
373       ) {
```

```
374             if (dataStore.getUint(Keys.ZFP_PROFIT_EXTRACT_OPTIMAL_ROI_0) > value) {
375                 revert Errors.ConfigValueExceedsAllowedRange(baseKey, value);
376             }
377         }
378         if (
379             baseKey == Keys.ZFP_PROFIT_EXTRACT_OPTIMAL_ROI_2
380         ) {
381             if (dataStore.getUint(Keys.ZFP_PROFIT_EXTRACT_OPTIMAL_ROI_1) > value) {
382                 revert Errors.ConfigValueExceedsAllowedRange(baseKey, value);
383             }
384         }
385         if (
386             baseKey == Keys.ZFP_PROFIT_EXTRACT_OPTIMAL_ROI_3
387         ) {
388             if (dataStore.getUint(Keys.ZFP_PROFIT_EXTRACT_OPTIMAL_ROI_2) > value) {
389                 revert Errors.ConfigValueExceedsAllowedRange(baseKey, value);
390             }
391         }
392         if (
393             baseKey == Keys.ZFP_PROFIT_EXTRACT_OPTIMAL_ROI_4
394         ) {
395             if (dataStore.getUint(Keys.ZFP_PROFIT_EXTRACT_OPTIMAL_ROI_3) > value) {
396                 revert Errors.ConfigValueExceedsAllowedRange(baseKey, value);
397             }
398         }
```

Listing 2.13: contracts/config/ConfigUtils.sol

3. It is recommended to add validation to ensure that the value of `MIN_ZFP_COLLATERAL_FACTOR` is greater than a minimum threshold (e.g., 0.18%). Moreover, when setting `MAX_ZFP_COLLATERAL_FACTOR`, it is recommended to add a non-zero check for the value of `MIN_ZFP_COLLATERAL_FACTOR`, ensuring that the value of `MIN_ZFP_COLLATERAL_FACTOR` is already configured.

```
406         if (
407             baseKey == Keys.MAX_ZFP_COLLATERAL_FACTOR
408         ) {
409             if (dataStore.getUint(Keys.MIN_ZFP_COLLATERAL_FACTOR) > value) {
410                 revert Errors.ConfigValueExceedsAllowedRange(baseKey, value);
411             }
412         }
```

Listing 2.14: contracts/config/ConfigUtils.sol

4. It is recommended to add validation for the values K and B to ensure the computed keep-ratio is strictly monotonically increasing.

5. The value of `MAX_PNL_FACTOR_FOR_ADL` should always be less or equal than the value of `MAX_PNL_FACTOR_FOR_WITHDRAWALS` to prevent the LP withdrawals being blocked while ADL is not triggered.

6. The value of `MAX_PNL_FACTOR_FOR_WITHDRAWALS` should always be less or equal than the value of `MAX_PNL_FACTOR_FOR_DEPOSITS` to prevent cross market arbitrage via deposit HLV in one market and withdraw in another market.

Suggestion Revise or add corresponding checks in the function `validateRange()`.

2.2.2 Revise the market token symbol

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the contract `MarketToken`, the constructor currently hardcodes the symbol `GM`. It is recommended to revise the market token symbol to match the protocol name.

```
12    constructor(RoleStore _roleStore, DataStore _dataStore) ERC20("HF Market", "GM") Bank(
          _roleStore, _dataStore) {
13    }
```

Listing 2.15: contracts/market/MarketToken.sol

Suggestion Revise the market token symbol.

2.3 Note

2.3.1 Proper position size delta configuration when applying ADL operations to ZFP positions

Introduced by `Version 1`

Description In the protocol, normal positions undergoing partial ADL trigger a collateral check via the function `willPositionCollateralBeSufficient()`, which enforces a minimum collateral bound. Therefore, partial ADL for normal positions does not fail due to leverage constraints. As for ZFP positions, a leverage range constraint (via the `willZFPPositionCollateralBeSufficient()` function) is applied during partial ADL, which includes the `willBeTooMuch` check that reverts when `remainingCollateralUsd >= maxZFPCollateralFactor * remainingSize`. Since ADL orders set `initialCollateralDeltaAmount` to `0`, the remaining collateral stays equal to the original amount while remaining position size decreases. However, this design may lead to unexpected failures for partial ADL on ZFP positions. For example, a ZFP position at 150X leverage (i.e., $100 in collateral and $15,000 in position size) with a partial ADL of $7,500 in position size results in a lower `maxCollateralUsd` (i.e., `1.2% * 7,500 = 90 < 100`), causing the ADL to revert. The project must ensure that the position size delta is properly configured when applying ADL to ZFP positions.

2.3.2 Proper priority between EFL and ADL operations

Introduced by `Version 1`

Description In the contract `EarlyForceLiquidationUtils`, the `validateEarlyForceLiquidation()` function does not distinguish between position types (i.e., EFL applies to both Normal and ZFP positions). EFL and ADL operate as independent atomic paths with different trigger dimensions.

The EFL operations target all positions. It always fully closes the position with no slippage protection and no post-execution check. As for the ADL mechanism, it aims at markets. Specifically, it is triggered when the condition `pnlToPoolFactor >= MAX_PNL_FACTOR_FOR_ADL` is realized by `ADL_KEEPER` and requires that `nextPnlToPoolFactor < previousPnlToPoolFactor`

after execution. Compared to ADL operations, which intend to improve pool solvency, EFL operations always fully close the position even if profit only marginally exceeds `maxProfitFactor` and has no post-execution solvency constraint.

However, there is no explicit priority between these two operations, which may lead to different outcomes. If EFL operations execute first, subsequent ADL operations may find the value of `pnlToPoolFactor` already reduced and skip execution. In contrast, if an ADL operation is executed first, the check (via the function `validateEarlyForceLiquidation()`) in the EFL operation may revert because the remaining position no longer meets the `maxProfitFactor` threshold. Therefore, the project must enforce a proper priority between EFL and ADL operations in the keepers' logic to avoid unexpected outcomes.

2.3.3 Risks for supporting RWA and commodity markets

Introduced by `Version 1`

Description If the project intends to support RWA and commodity markets that require on-hours/off-hours parameter switching, the `CONFIG_KEEPER` role must properly shift risk parameters for related markets between on-hours and off-hours configurations based on schedule.

Since existing positions remain open when off-hours begin, any tightening of liquidation thresholds during the transition affects all open positions. ZFP positions at 75x-555x leverage are extremely sensitive to such threshold adjustments and a small tightening of the variable `minZFPCollateralFactorForLiquidation` could instantly push positions that were previously safe into liquidable status.

2.3.4 Potential HLV price inflation due to EFL operations

Introduced by `Version 1`

Description The HLV token pricing in the function `_getHlvMarketValue()` calls the function `getPoolValueInfo()` with `MAX_PNL_FACTOR_FOR_DEPOSITS` to calculate PnL, which in turn calls the function `getPnl()`, summing all positions including ZFP positions without applying the parameter `MAX_PROFIT_FACTOR`. However, at position settlement, the function `getPositionPnlUsd()` applies the parameter `MAX_PROFIT_FACTOR`, truncating actual payout. This difference inflates the PnL deduction from pool value and depresses HLV token pricing. Therefore, when a high-profitable ZFP position is closed via EFL and the inflated PnL deduction is removed, HLV token price jumps upward, creating an arbitrage opportunity.

```
480   function getPnl(
481       DataStore dataStore,
482       Market.Props memory market,
483       Price.Props memory indexTokenPrice,
484       bool isLong,
485       bool maximize
486   ) internal view returns (int256) {
487       int256 openInterest = getOpenInterest(dataStore, market, isLong).toInt256();
488       uint256 openInterestInTokens = getOpenInterestInTokens(dataStore, market, isLong);
489       if (openInterest == 0 || openInterestInTokens == 0) {
490           return 0;
```

```
491         }
492
493         uint256 price = indexTokenPrice.pickPriceForPnl(isLong, maximize);
494
495         // openInterest is the cost of all positions, openInterestValue is the current worth of all
                positions
496         int256 openInterestValue = (openInterestInTokens * price).toInt256();
497         int256 pnl = isLong ? openInterestValue - openInterest : openInterest - openInterestValue;
498
499         return pnl;
500     }
```

Listing 2.16: contracts/market/MarketUtils.sol

```
238         // cap positionPnlUsd by maxProfitFactor * collateralUsd
239         if (cache.positionPnlUsd > 0) {
240             uint256 maxProfitFactor = MarketUtils.getMaxProfitFactor(dataStore, market.marketToken)
                    ;
241             if (maxProfitFactor > 0) {
242                 Price.Props memory collateralTokenPrice = position.collateralToken() == market.
                        longToken
243                     ? prices.longTokenPrice
244                     : prices.shortTokenPrice;
245                 uint256 maxProfitUsd = Precision.applyFactor(position.collateralAmount() *
                        collateralTokenPrice.min, maxProfitFactor);
246                 if (sizeDeltaUsd < position.sizeInUsd()) {
247                     maxProfitUsd = maxProfitUsd * sizeDeltaUsd / position.sizeInUsd();
248                 }
249                 if (cache.positionPnlUsd > maxProfitUsd.toInt256()) {
250                     cache.positionPnlUsd = maxProfitUsd.toInt256();
251                 }
252             }
253         }
254
255         return (cache.positionPnlUsd, cache.uncappedPositionPnlUsd, cache.sizeDeltaInTokens);
```

Listing 2.17: contracts/position/PositionUtils.sol

2.3.5 Ensure gasless order creation entry points are disabled

Introduced by Version 1

Description In the contract `BaseGelatoRelayRouter`, the function `_validateGaslessFeature()` checks whether the gasless feature is disabled via the `GASLESS_FEATURE_DISABLED` key. If this flag is not set to `true`, relay-based order creation entry points remain active. The EIP-712 struct hash computed in `RelayUtils._getCreateOrderParamsStructHash` does not include `orderPositionTy`. Thus, a malicious relayer could alter a user's position type between Normal and ZFP without invalidating the signature. Since the protocol is deployed only on BSC and does not use Gelato Relay, these gasless entry points must be disabled to eliminate this attack surface.

```
419     function _validateGaslessFeature() internal view {
420         FeatureUtils.validateFeature(dataStore, Keys.gaslessFeatureDisabledKey(address(this)));
```

```
421     }
```

Listing 2.18: contracts/router/relay/BaseGelatoRelayRouter.sol

```
495     function _getCreateOrderParamsStructHash(
496         IBaseOrderUtils.CreateOrderParams calldata params
497     ) private pure returns (bytes32) {
498         return
499             keccak256(
500                 abi.encode(
501                     CREATE_ORDER_PARAMS_TYPEHASH,
502                     _getCreateOrderAddressesStructHash(params.addresses),
503                     _getCreateOrderNumbersStructHash(params.numbers),
504                     uint256(params.orderType),
505                     uint256(params.decreasePositionSwapType),
506                     params.isLong,
507                     params.shouldUnwrapNativeToken,
508                     params.autoCancel,
509                     params.referralCode,
510                     keccak256(abi.encodePacked(params.dataList))
511                 )
512             );
513     }
```

Listing 2.19: contracts/router/relay/RelayUtils.sol

2.3.6 Potential centralization risks

Introduced by Version 1

Description In this project, several privileged roles (e.g., ORDER_KEEPER) can conduct sensitive operations, which introduces potential centralization risks. For example, the ORDER_KEEPER role controls the execution of all user-submitted orders, deposits, withdrawals, and shifts. A malicious or compromised ORDER_KEEPER could selectively execute orders, front-running user orders for profit, censoring time-sensitive orders such as stop-losses during volatile market conditions, and manipulating execution orders to extract value through ordering arbitrage. If the private keys of the privileged accounts are lost or maliciously exploited, it could pose a significant risk to the protocol.

2.3.7 Deprecated modules and logic

Introduced by Version 1

Description The following modules are inactive, deprecated, unused, and out of scope for this audit.

1. Multichain Module (i.e., contracts/multichain)
2. Swap Module (i.e., contracts/swap)
3. Fee Module (i.e., contracts/fee/FeeDistributor*.sol, contracts/fee/FeeBatch*.sol, and contracts/fee/FeeSwapUtils.sol)


BOOST WEB3 THROUGH NEXT-GENERATION SECURITY & USABILITY INNOVATIONS